**Hướng dẫn tạo và sử dụng ThreadPool trong Java**

*Đăng vào 27/02/2018 . Được đăng bởi*[**GP Coder**](https://gpcoder.com/author/gpadmin/)*.****17783****Lượt xem . Toàn màn hình*

**Nội dung**[[Ẩn](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/)]

* [1 Giới thiệu ThreadPool](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/#Gioi_thieu_ThreadPool)
* [2 Executor là gì?](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/#Executor_la_gi)
* [3 Các ví dụ minh họa tạo và sử dụng ThreadPool](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/#Cac_vi_du_minh_hoa_tao_va_su_dung_ThreadPool)
  + [3.1 Tạo WorkerThread](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/#TaoWorkerThread)
  + [3.2 Ví dụ sử dụng newSingleThreadExecutor()](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/#Vi_du_su_dung_newSingleThreadExecutor)
  + [3.3 Ví dụ sử dụng newFixedThreadPool()](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/#Vi_du_su_dungnewFixedThreadPool)
  + [3.4 Sử dụng newCachedThreadPool()](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/#Su_dungnewCachedThreadPool)
  + [3.5 Sử dụng newScheduledThreadPool()](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/#Su_dungnewScheduledThreadPool)
* [4 Custom Thread Pool Executor](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/#Custom_Thread_Pool_Executor)
  + [4.1 Ví dụ minh họa](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/#Vi_du_minh_hoa)
* [5 Một vài lưu ý về sử dụng ExecutorService](https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/#Mot_vai_luu_y_ve_su_dungExecutorService)

**Giới thiệu ThreadPool**

Xét về hiệu suất, tạo ra một Thread mới là một hoạt động tốn kém bởi vì nó đòi hỏi hệ điều hành cung cấp tài nguyên để có thể thực thi task (tác vụ). Trên thực tế, **ThreadPool** được sử dụng cho các ứng dụng quy mô lớn khởi chạy rất nhiều luồng ngắn ngủi để sử dụng hiệu quả các tài nguyên và tăng hiệu suất.

Trong Java, ThreadPool được dùng để giới hạn số lượng Thread được chạy bên trong ứng dụng của chúng ta trong cùng một thời điểm. Nếu chúng ta không có sự giới hạn này, mỗi khi có một Thread mới được tạo ra và được cấp phát bộ nhớ bằng từ khóa new thì sẽ có vấn đề về bộ nhớ và hiệu suất, có thể dẫn đến lỗi crash chương trình.

**Ví dụ:** Khi chúng ta viết chương trình tải các tập tin từ Internet, mỗi tập tin cần 1 Thread để thực hiện quá trình tải, giả sử cần tải 100 tệp hình ảnh thì chúng ta phải cần tới 100 Thread hoạt động cùng một thời điểm trong cùng một chương trình. Điều này sẽ dễ dẫn đến lỗi quá tải của chương trình, làm ảnh hưởng đến hiệu suất và có thể dẫn đến gây lỗi (crash) chương trình.

Vì vậy, thay vì tạo các luồng mới khi các task (nhiệm vụ) mới đến, một ThreadPool sẽ giữ một số luồng nhàn rỗi (no task) đã sẵn sàng để thực hiện tác vụ nếu cần. Sau khi một thread hoàn thành việc thực thi một tác vụ, nó sẽ không chết. Thay vào đó nó vẫn không hoạt động trong ThreadPool và chờ đợi được lựa chọn để thực hiện nhiệm vụ mới.

Chúng ta có thể giới hạn một số lượng nhất định các Thread đồng thời trong ThreadPool, rất hữu ích để ngăn chặn quá tải. Nếu tất cả các Thread đang bận rộn thực hiện nhiệm vụ, nhiệm vụ mới được đặt trong một hàng đợi (**BlockingQueue**), chờ đợi một Thread trở nên có sẵn.

[Diagram

Description automatically generated](https://gpcoder.com/wp-content/uploads/2018/02/threadpool-executor.png)

Java Concurrency API hỗ trợ một vài loại ThreadPool sau:

* **Cached thread pool**: giữ một số luồng còn sống (alive) và tạo ra các luồng mới nếu cần.
* **Fixed thread pool**: giới hạn số lượng tối đa của các Thread được tạo ra để thực thi các task (nhiệm vụ). Các task khác đang chờ trong hàng đợi (BlockingQueue).
* **Single-threaded pool**: chỉ giữ một Thread thực thi một nhiệm vụ một lúc.
* **Fork/Join pool**: một Thread đặc biệt sử dụng Fork/ Join Framework để tận dụng lợi thế của nhiều bộ vi xử lý để thực hiện công việc lớn nhanh hơn bằng cách chia nhỏ công việc thành các phần nhỏ hơn để xử lý đệ quy.

Trong thực tế, ThreadPool được sử dụng rộng rãi trong các máy chủ web, nơi một ThreadPool được sử dụng để phục vụ các yêu cầu của khách hàng. Thread pool cũng được sử dụng trong các ứng dụng cơ sở dữ liệu nơi mà một ThreadPool được sử dụng để duy trì các kết nối mở với cơ sở dữ liệu.

Việc cài đặt ThreadPool là một công việc phức tạp, nhưng chúng ta không cần phải lo lắng điều này bởi vì Java Concurrency API đã xây dựng sẵn (build-in) các lớp hỗ trợ ThreadPool trong gói ***java.util.concurrent***. Chúng ta sẽ tiếp tục tìm hiểu ở các phần tiếp theo của bài viết này.

**Executor là gì?**

Một Executor là một đối tượng chịu trách nhiệm quản lý các luồng và thực hiện các tác vụ Runnable được yêu cầu xử lý. Nó tách riêng các chi tiết của việc tạo Thread, lập kế hoạch (scheduling), … để chúng ta có thể tập trung phát triển logic của tác vụ mà không quan tâm đến các chi tiết quản lý Thread.

[A picture containing diagram

Description automatically generated](https://gpcoder.com/wp-content/uploads/2018/02/threadpool-executor-service.png)

Java Concurrency API định nghĩa 3 interfaces cơ bản sau cho các Executor:

* **Executor**: là interface cha của tất cả Executor. Nó xác định chỉ một phương thực **excute(Runnable)**.
* **ExecutorService**: là một Executor cho phép theo dõi tiến trình của các tác vụ trả về giá trị (**Callable**) thông qua đối tượng **Future**, và quản lý việc kết thúc các luồng. Các phương thức chính của nó bao gồm **submit()** và **shutdown()**.
* **ScheduledExecutorService**: là một ExecutorService có thể lên lịch cho các tác vụ để thực thi sau một khoảng thời gian nhất định, hoặc để thực hiện định kỳ. Các phương thức chính của nó là **schedule()**, **scheduleAtFixedRate()** and **scheduleWithFixedDelay()**.

Chúng có thể tạo một Executor bằng cách sử dụng một trong các phương thức được cung cấp bởi lớp tiện ích **Executors** như sau:

* **newSingleThreadExecutor()**: trong ThreadPool chỉ có 1 Thread và các task (nhiệm vụ) sẽ được xử lý một cách tuần tự.
* **newCachedThreadPool()**: trong ThreadPool sẽ có nhiều Thread và các nhiệm vụ sẽ được xử lý một cách song song. Các Thread cũ sau khi xử lý xong sẽ được sử dụng lại cho nhiệm vụ mới. Mặc định nếu một Thread không được sử dụng trong vòng 60 giây thì Thread đó sẽ bị tắt.
* **newFixedThreadPool(int n)**: trong ThreadPool sẽ được cố định các Thread. Nếu một nhiệm vụ mới được đưa vào mà các Thread đều đang “bận rộn” thì nhiệm vụ đó sẽ được gửi vào Blocking Queue và sau đó nếu có một Thread đã thực thi xong nhiệm vụ của nó thì nhiệm vụ đang ở trong Queue đó sẽ được push ra khỏi Queue và được Thread đó xử lý tiếp.
* **newScheduledThreadPool(int corePoolSize)**: tương tự như newCachedThreadPool() nhưng sẽ có thời gian delay giữa các Thread.
* **newSingleThreadScheduledExecutor()**: tương tự như newSingleThreadExecutor() nhưng sẽ có khoảng thời gian delay giữa các Thread.

**Các ví dụ minh họa tạo và sử dụng ThreadPool**

**Tạo WorkerThread**

Trước khi đi vào chi tiết cách sử dụng các phương thức của lớp Executors, hãy xem một task (tác vụ) sẽ được sử dụng để minh họa cho các ví dụ tiếp theo.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | **package** com.gpcoder.threadpool;    **public** **class** WorkerThread **implements** Runnable {    **private** String task;    **public** WorkerThread(String s) {  **this**.task = s;      }        @Override  **public** **void** run() {          System.out.println(Thread.currentThread().getName() + " Starting. Task = " + task);          processCommand();          System.out.println(Thread.currentThread().getName() + " Finished.");      }    **private** **void** processCommand() {  **try** {              Thread.sleep(2000);          } **catch** (InterruptedException e) {              e.printStackTrace();          }      }  } |

**Ví dụ sử dụng newSingleThreadExecutor()**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | **package** com.gpcoder.threadpool;    **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;    **public** **class** SingleThreadExecutorExample {  **public** **static** **void** main(String[] args) {          ExecutorService executor = Executors.newSingleThreadExecutor();    **for** (**int** i = 1; i <= 10; i++) {              Runnable worker = **new** WorkerThread("" + i);              executor.execute(worker);          }            /\*           \* Initiates an orderly shutdown in which previously submitted tasks are           \* executed, but no new tasks will be accepted. Invocation has no additional           \* effect if already shut down. This method does not wait for previously           \* submitted tasks to complete execution. Use awaitTermination to do that.           \*/          executor.shutdown();            // Wait until all threads are finish  **while** (!executor.isTerminated()) {              // Running ...          }            System.out.println("Finished all threads");      }  } |

Trong chương trình trên, tôi đã tạo ra ThreadPool sử dụng phương thức newSingleThreadExecutor() vì vậy kích thước của ThreadPool là 1, nên nó sẽ bắt đầu thực thi chương trình trên với 1 task và các task khác sẽ ở trạng thái đợi (waiting), ngay khi một task hoàn thành, một task khác từ hàng đợi sẽ được chọn và thực thi. Ta có kết quả của chương trình trên như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | pool-1-thread-1 Starting. Task = 1  pool-1-thread-1 Finished.  pool-1-thread-1 Starting. Task = 2  pool-1-thread-1 Finished.  pool-1-thread-1 Starting. Task = 3  pool-1-thread-1 Finished.  pool-1-thread-1 Starting. Task = 4  pool-1-thread-1 Finished.  pool-1-thread-1 Starting. Task = 5  pool-1-thread-1 Finished.  pool-1-thread-1 Starting. Task = 6  pool-1-thread-1 Finished.  pool-1-thread-1 Starting. Task = 7  pool-1-thread-1 Finished.  pool-1-thread-1 Starting. Task = 8  pool-1-thread-1 Finished.  pool-1-thread-1 Starting. Task = 9  pool-1-thread-1 Finished.  pool-1-thread-1 Starting. Task = 10  pool-1-thread-1 Finished.  Finished all threads |

**Ví dụ sử dụng newFixedThreadPool()**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | **package** com.gpcoder.threadpool;    **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;    **public** **class** FixedThreadPoolExample {    **public** **static** **final** **int** NUM\_OF\_THREAD = 5;    **public** **static** **void** main(String[] args) {            ExecutorService executor = Executors.newFixedThreadPool(NUM\_OF\_THREAD);    **for** (**int** i = 1; i <= 10; i++) {              Runnable worker = **new** WorkerThread("" + i);              executor.execute(worker);          }            /\*           \* Initiates an orderly shutdown in which previously submitted tasks are           \* executed, but no new tasks will be accepted. Invocation has no additional           \* effect if already shut down. This method does not wait for previously           \* submitted tasks to complete execution. Use awaitTermination to do that.           \*/          executor.shutdown();            // Wait until all threads are finish  **while** (!executor.isTerminated()) {              // Running ...          }            System.out.println("Finished all threads");      }  } |

Trong chương trình trên, tôi đã tạo ra ThreadPool có kích thước cố định là 5. Sau đó, tôi đã tạo 10 task (công việc) vào ThreadPool, vì kích thước ThreadPool là 5, nên nó sẽ bắt đầu thực thi chương trình trên vói 5 task và các task khác sẽ ở trạng thái đợi (waiting), ngay khi một task hoàn thành, một task khác từ hàng đợi sẽ được chọn và thực thi. Ta có kết quả của chương trình trên như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | pool-1-thread-3 Starting. Task = 3  pool-1-thread-2 Starting. Task = 2  pool-1-thread-1 Starting. Task = 1  pool-1-thread-4 Starting. Task = 4  pool-1-thread-5 Starting. Task = 5  pool-1-thread-2 Finished.  pool-1-thread-3 Finished.  pool-1-thread-2 Starting. Task = 6  pool-1-thread-3 Starting. Task = 7  pool-1-thread-1 Finished.  pool-1-thread-5 Finished.  pool-1-thread-4 Finished.  pool-1-thread-5 Starting. Task = 9  pool-1-thread-1 Starting. Task = 8  pool-1-thread-4 Starting. Task = 10  pool-1-thread-3 Finished.  pool-1-thread-2 Finished.  pool-1-thread-5 Finished.  pool-1-thread-1 Finished.  pool-1-thread-4 Finished.  Finished all threads |

**Sử dụng newCachedThreadPool()**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | **package** com.gpcoder.threadpool;    **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;    **public** **class** CachedThreadPoolExample {  **public** **static** **void** main(String[] args) **throws** InterruptedException {          ExecutorService executor = Executors.newCachedThreadPool();    **for** (**int** i = 1; i <= 10; i++) {              Runnable worker = **new** WorkerThread("" + i);              executor.execute(worker);              Thread.sleep(400);          }            /\*           \* Initiates an orderly shutdown in which previously submitted tasks are           \* executed, but no new tasks will be accepted. Invocation has no additional           \* effect if already shut down. This method does not wait for previously           \* submitted tasks to complete execution. Use awaitTermination to do that.           \*/          executor.shutdown();            // Wait until all threads are finish  **while** (!executor.isTerminated()) {              // Running ...          }            System.out.println("Finished all threads");      }  } |

Trong chương trình trên, tôi đã tạo ra ThreadPool sử dụng phương thức newCachedThreadPool() vì vậy kích thước của ThreadPool là gần như không giới hạn (Integer.MAX\_VALUE), nên nó sẽ bắt đầu thực thi chương trình trên với 1 task và các task khác sẽ ở không phải đợi. Nếu có Thread rãnh thì nó sẽ nhận task và thực thi. Nếu không có Thread rãnh thì nó sẽ tạo một Thread mới và thực thi. Ta có kết quả của chương trình trên như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | pool-1-thread-1 Starting. Task = 1  pool-1-thread-2 Starting. Task = 2  pool-1-thread-3 Starting. Task = 3  pool-1-thread-4 Starting. Task = 4  pool-1-thread-5 Starting. Task = 5  pool-1-thread-1 Finished.  pool-1-thread-6 Starting. Task = 6  pool-1-thread-1 Starting. Task = 7  pool-1-thread-2 Finished.  pool-1-thread-3 Finished.  pool-1-thread-3 Starting. Task = 8  pool-1-thread-4 Finished.  pool-1-thread-4 Starting. Task = 9  pool-1-thread-5 Finished.  pool-1-thread-5 Starting. Task = 10  pool-1-thread-6 Finished.  pool-1-thread-1 Finished.  pool-1-thread-3 Finished.  pool-1-thread-4 Finished.  pool-1-thread-5 Finished.  Finished all threads |

**Sử dụng newScheduledThreadPool()**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | **package** com.gpcoder.threadpool;    **import** java.util.concurrent.Executors;  **import** java.util.concurrent.ScheduledExecutorService;  **import** java.util.concurrent.TimeUnit;    **public** **class** ScheduledThreadPoolExample {    **public** **static** **final** **int** NUM\_OF\_THREAD = 2;  **public** **static** **final** **int** INITIAL\_DELAY = 1; // second  **public** **static** **final** **int** DELAY = 3; // second    **public** **static** **void** main(String[] args) **throws** InterruptedException {          ScheduledExecutorService executor = Executors.newScheduledThreadPool(NUM\_OF\_THREAD);    **for** (**int** i = 1; i <= 5; i++) {              Runnable worker = **new** WorkerThread("" + i);              executor.scheduleWithFixedDelay(worker, INITIAL\_DELAY, DELAY, TimeUnit.SECONDS);          }            // waits for termination for 10 seconds only          executor.awaitTermination(10, TimeUnit.SECONDS);            /\*           \* Initiates an orderly shutdown in which previously submitted tasks are           \* executed, but no new tasks will be accepted. Invocation has no additional           \* effect if already shut down. This method does not wait for previously           \* submitted tasks to complete execution. Use awaitTermination to do that.           \*/          executor.shutdown();            // Wait until all threads are finish  **while** (!executor.isTerminated()) {              // Running ...          }            System.out.println("Finished all threads");      }  } |

Trong chương trình trên, tôi đã tạo ra ThreadPool có kích thước cố định là 2. Sau đó, tôi đã tạo 5 task (công việc) vào ThreadPool, vì kích thước ThreadPool là 2, nên nó sẽ bắt đầu thực thi chương trình trên vói 5 task và các task khác sẽ ở trạng thái đợi (waiting), ngay khi một task hoàn thành, một task khác từ hàng đợi sẽ được chọn và thực thi.

Các thread này được lên kế hoạch thực thi bằng phương thức **scheduleWithFixedDelay()**, sau khi thực thi lần đầu tiên, nó sẽ được gọi để thực thi lại sau khoảng thời gian delay xác định. Phương thức này gồm 4 tham số như sau:

* **Runnable**: tác vụ sẽ được thực thi.
* **initialDelay**: thời gian trì hoãn trước khi thực thi lần đầu tiên.
* **delay**: thời gian trì hoãn kể từ khi thực thi lần trước đó kết thúc.
* **TimeUnit**: đơn vị thời gian của tham số initialDelay và delay.

Trong chương trình trên, tôi còn sử dụng một phương thức **awaitTermination()**: phương thức này xác định thời gian chờ tối đa để các task hoàn thành thực thi.

Ta có kết quả của chương trình trên như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | pool-1-thread-1 Starting. Task = 1  pool-1-thread-2 Starting. Task = 2  pool-1-thread-1 Finished.  pool-1-thread-2 Finished.  pool-1-thread-1 Starting. Task = 3  pool-1-thread-2 Starting. Task = 4  pool-1-thread-1 Finished.  pool-1-thread-2 Finished.  pool-1-thread-1 Starting. Task = 5  pool-1-thread-2 Starting. Task = 1  pool-1-thread-1 Finished.  pool-1-thread-1 Starting. Task = 2  pool-1-thread-2 Finished.  pool-1-thread-2 Starting. Task = 3  pool-1-thread-1 Finished.  pool-1-thread-1 Starting. Task = 4  pool-1-thread-2 Finished.  pool-1-thread-1 Finished.  Finished all threads |

**Custom Thread Pool Executor**

Trong trường hợp chúng ta muốn kiểm soát nhiều hơn các hành vi của một ThreadPool, chúng ta có thể tạo một bộ xử lý ThreadPool trực tiếp từ lớp **ThreadPoolExecutor** thay vì các phương thức có sẵn của lớp tiện ích **Executors**.

Một ThreadPoolExecutor có hàm khởi tạo như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **public** ThreadPoolExecutor(**int** corePoolSize,  **int** maximumPoolSize,  **long** keepAliveTime,  TimeUnit unit,  BlockingQueue<Runnable> workQueue,  ThreadFactory threadFactory,  RejectedExecutionHandler handler) |

Các tham số của phương thức này là:

* **int corePoolSize**: số lượng các Thread để giữ trong ThreadPool.
* **int maximumPoolSize**: số lượng tối đa của Thread để cho phép trong ThreadPool.
* **long keepAliveTime**: nếu ThreadPool hiện có nhiều hơn các Thread corePoolSize, Thread vượt quá sẽ bị chấm dứt nếu nó đã được nhàn rỗi hơn một khoảng thời gian keepAliveTime.
* **TimeUnit unit**: đơn vị thời gian cho đối số keepAliveTime. Có thể là NANOSECONDS, MILLISECONDS, SECONDS, MINUTES, HOURS AND DAYS.
* **BlockingQueue workQueue**: hàng đợi được sử dụng để giữ các task trước khi chúng được thực hiện. Các lựa chọn mặc định là SynchronousQueue cho multi-thread và LinkedBlockingQueue cho các single-thread.
* **ThreadFactory threadFactory**: factory được sử dụng khi tạo một Thread mới.
* **RejectedExecutionHandler handler**: handler được sử dụng khi một execute bị chặn (blocked) khi vượt quá maximumPoolSize.

**Ví dụ minh họa**

Cài đặt **RejectedExecutionHandler** để xử lý các tác vụ bị từ chối.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **package** com.gpcoder.threadpool.custom;    **import** java.util.concurrent.RejectedExecutionHandler;  **import** java.util.concurrent.ThreadPoolExecutor;    **public** **class** RejectedExecutionHandlerImpl **implements** RejectedExecutionHandler {        @Override  **public** **void** rejectedExecution(Runnable r, ThreadPoolExecutor executor) {          System.out.println(r.toString() + " is rejected");      }    } |

Tạo một lớp ThreadFactoryBuilder để tùy chỉnh ThreadFactory trong việc đặt tên Thread, xử lý ngoại lệ, xác định độ ưu tiên của Thread, xác định luồng daemon:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94 | **package** com.gpcoder.threadpool.custom;    **import** java.lang.Thread.UncaughtExceptionHandler;  **import** java.util.concurrent.Executors;  **import** java.util.concurrent.ThreadFactory;  **import** java.util.concurrent.atomic.AtomicLong;    **public** **class** ThreadFactoryBuilder {  **private** String namePrefix = **null**;  **private** **boolean** daemon = **false**;  **private** **int** priority = Thread.NORM\_PRIORITY;  **private** ThreadFactory backingThreadFactory = **null**;  **private** UncaughtExceptionHandler uncaughtExceptionHandler = **null**;    **public** ThreadFactoryBuilder() {        }  **public** ThreadFactoryBuilder setNamePrefix(String namePrefix) {  **if** (namePrefix == **null**) {  **throw** **new** NullPointerException();          }  **this**.namePrefix = namePrefix;  **return** **this**;      }    **public** ThreadFactoryBuilder setDaemon(**boolean** daemon) {  **this**.daemon = daemon;  **return** **this**;      }    **public** ThreadFactoryBuilder setPriority(**int** priority) {  **if** (priority < Thread.MIN\_PRIORITY) { **throw** **new** IllegalArgumentException( String.format("Thread priority (%s) must be >= %s", priority, Thread.MIN\_PRIORITY));          }    **if** (priority > Thread.MAX\_PRIORITY) {  **throw** **new** IllegalArgumentException(                      String.format("Thread priority (%s) must be <= %s", priority, Thread.MAX\_PRIORITY));          }    **this**.priority = priority;  **return** **this**;      }    **public** ThreadFactoryBuilder setUncaughtExceptionHandler(UncaughtExceptionHandler uncaughtExceptionHandler) {  **if** (**null** == uncaughtExceptionHandler) {  **throw** **new** NullPointerException("UncaughtExceptionHandler cannot be null");          }  **this**.uncaughtExceptionHandler = uncaughtExceptionHandler;  **return** **this**;      }    **public** ThreadFactoryBuilder setThreadFactory(ThreadFactory backingThreadFactory) {  **if** (**null** == uncaughtExceptionHandler) {  **throw** **new** NullPointerException("BackingThreadFactory cannot be null");          }  **this**.backingThreadFactory = backingThreadFactory;  **return** **this**;      }    **public** ThreadFactory build() {  **return** build(**this**);      }    **private** **static** ThreadFactory build(ThreadFactoryBuilder builder) {  **final** String namePrefix = builder.namePrefix;  **final** Boolean daemon = builder.daemon;  **final** Integer priority = builder.priority;  **final** UncaughtExceptionHandler uncaughtExceptionHandler = builder.uncaughtExceptionHandler;  **final** ThreadFactory backingThreadFactory = (builder.backingThreadFactory != **null**) ? builder.backingThreadFactory                  : Executors.defaultThreadFactory();    **final** AtomicLong count = **new** AtomicLong(0);    **return** **new** ThreadFactory() {              @Override  **public** Thread newThread(Runnable runnable) {                  Thread thread = backingThreadFactory.newThread(runnable);  **if** (namePrefix != **null**) {                      thread.setName(namePrefix + "-" + count.getAndIncrement());                  }  **if** (daemon != **null**) {                      thread.setDaemon(daemon);                  }  **if** (priority != **null**) {                      thread.setPriority(priority);                  }  **if** (uncaughtExceptionHandler != **null**) {                      thread.setUncaughtExceptionHandler(uncaughtExceptionHandler);                  }  **return** thread;              }          };      }  } |

Tạo lớp MonitorThread để giám sát trạng thái của executor hiện tại, kích thước của ThreadPool, số lượng Thread đang hoạt động, …

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | **package** com.gpcoder.threadpool.custom;    **import** java.util.concurrent.ThreadPoolExecutor;    **public** **class** MonitorThread **implements** Runnable {  **private** ThreadPoolExecutor executor;  **private** **int** seconds;  **private** **boolean** run = **true**;    **public** MonitorThread(ThreadPoolExecutor executor, **int** delay) {  **this**.executor = executor;  **this**.seconds = delay;      }    **public** **void** shutdown() {  **this**.run = **false**;      }        @Override  **public** **void** run() {  **while** (run) {              System.out.println(String.format(                      "[monitor] [%d/%d] Active: %d, Completed: %d, Task: %d, isShutdown: %s, isTerminated: %s",  **this**.executor.getPoolSize(), **this**.executor.getCorePoolSize(), **this**.executor.getActiveCount(),  **this**.executor.getCompletedTaskCount(), **this**.executor.getTaskCount(), **this**.executor.isShutdown(),  **this**.executor.isTerminated()));  **try** {                  Thread.sleep(seconds \* 1000);              } **catch** (InterruptedException e) {                  e.printStackTrace();              }          }        }  } |

Tạo một lớp WorkerThread để mô phỏng các tác vụ của hệ thống:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | **package** com.gpcoder.threadpool.custom;    **public** **class** WorkerThread **implements** Runnable {    **private** String task;    **public** WorkerThread(String s) {  **this**.task = s;      }        @Override  **public** **void** run() {          System.out.println(Thread.currentThread().getName() + " Starting. Task = " + task);          processCommand();          System.out.println(Thread.currentThread().getName() + " Finished.");      }    **private** **void** processCommand() {  **try** {              Thread.sleep(5000);          } **catch** (InterruptedException e) {              e.printStackTrace();          }      }        @Override  **public** String toString() {  **return** "Task = " + task;      }  } |

Tạo chương trình minh họa thực thi tác vụ nhiều tác vụ WorkerThread, được giám sát bởi MonitorThread với các tùy chỉnh RejectedExecutionHandler, ThreadFactory.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | **package** com.gpcoder.threadpool.custom;    **import** java.lang.Thread.UncaughtExceptionHandler;  **import** java.util.concurrent.ArrayBlockingQueue;  **import** java.util.concurrent.BlockingQueue;  **import** java.util.concurrent.ThreadFactory;  **import** java.util.concurrent.ThreadPoolExecutor;  **import** java.util.concurrent.TimeUnit;    **public** **class** ThreadPoolExecutorExample {    **public** **static** **void** main(String args[]) **throws** InterruptedException {            // the number of threads to keep in the pool, even if they are idle  **final** **int** CORE\_POOL\_SIZE = 2; //            // the maximum number of threads to allow in the pool  **final** **int** MAX\_POOL\_SIZE = 4;            // the queue to use for holding tasks before they are executed  **final** **long** KEEP\_ALIVE\_TIME = 10;            // the queue to use for holding tasks before they are executed. This queue will          // hold only the Runnable tasks submitted by the execute method.          BlockingQueue<Runnable> workQueue = **new** ArrayBlockingQueue<Runnable>(2);            // RejectedExecutionHandler implementation          RejectedExecutionHandlerImpl rejectionHandler = **new** RejectedExecutionHandlerImpl();            // The handler to use when execution is blocked because the thread bounds          // and queue capacities are reached          // ThreadFactory threadFactory = Executors.defaultThreadFactory();          ThreadFactory threadFactory = **new** ThreadFactoryBuilder()                  .setNamePrefix("GPCoder-ThreadPool")                  .setDaemon(**false**)                  .setPriority(Thread.MAX\_PRIORITY)                  .setUncaughtExceptionHandler(**new** UncaughtExceptionHandler() {                      @Override  **public** **void** uncaughtException(Thread t, Throwable e) {                          System.err.println(String.format("Custom Exception: Thread %s threw exception - %s",                                  t.getName(), e.getMessage()));                        }                  }).build();            // creating the ThreadPoolExecutor          ThreadPoolExecutor executorPool = **new** ThreadPoolExecutor(                  CORE\_POOL\_SIZE, MAX\_POOL\_SIZE, KEEP\_ALIVE\_TIME,                  TimeUnit.SECONDS, workQueue, threadFactory, rejectionHandler);            // start the monitoring thread          MonitorThread monitor = **new** MonitorThread(executorPool, 3);          Thread monitorThread = **new** Thread(monitor);          monitorThread.start();            // submit work to the thread pool  **for** (**int** i = 1; i <= 10; i++) {              executorPool.execute(**new** WorkerThread("cmd" + i));          }            Thread.sleep(30000);            // shut down the pool          executorPool.shutdown();            // shut down the monitor thread          Thread.sleep(5000);          monitor.shutdown();        }  } |

Trong chương trình trên, tôi đã tạo ra ThreadPool sử dụng **ThreadPoolExecutor** với:

* Số lượng Thread ban đầu là 2 (**corePoolSize** ) và tối đa là 4 (**maximumPoolSize**) Thread trong ThreadPool.
* Một **workQueue**để lưu giữ các task sẽ được thực thi.
* Chương trình sẽ yêu cầu thực thi **10 task**.
* Do **maximumPoolSize**là 4 nên chỉ có 4 Thread được thực thi cùng lúc, **workQueue**là 2 nên sẽ có 2 task được nằm trong hàng đợi (BlockingQueue), 4 Thread còn lại sẽ bị từ chối thực thi.
* Sau khi Thread hoàn thành thực thi task một khoảng thời gian **keepAliveTime**, nếu số lượng Thread trong ThreadPool lớn hơn corePoolSize và workQueue không còn task yêu cầu thực thi thì Thread sẽ bị đóng lại (đóng các Thread cho đến khi số lượng Thread = corePoolSize  trong ThreadPool).
* ThreadFactory sẽ tạo tên Thread với tiếp đầu ngữ là là **GPCoder-ThreadPool**.
* **MonitorThread** định kỳ khoảng 3 giây sẽ hiển thị thông tin của executor hiện tại.

Hãy xem kết quả thực thi chương trình trên như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | GPCoder-ThreadPool-0 Starting. Task = cmd1  GPCoder-ThreadPool-1 Starting. Task = cmd2  GPCoder-ThreadPool-2 Starting. Task = cmd5  GPCoder-ThreadPool-3 Starting. Task = cmd6  Task = cmd7 is rejected  Task = cmd8 is rejected  Task = cmd9 is rejected  Task = cmd10 is rejected  [monitor] [0/2] Active: 0, Completed: 0, Task: 2, isShutdown: **false**, isTerminated: **false**  [monitor] [4/2] Active: 4, Completed: 0, Task: 6, isShutdown: **false**, isTerminated: **false**  GPCoder-ThreadPool-0 Finished.  GPCoder-ThreadPool-1 Finished.  GPCoder-ThreadPool-0 Starting. Task = cmd3  GPCoder-ThreadPool-1 Starting. Task = cmd4  GPCoder-ThreadPool-2 Finished.  GPCoder-ThreadPool-3 Finished.  [monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: **false**, isTerminated: **false**  [monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: **false**, isTerminated: **false**  GPCoder-ThreadPool-0 Finished.  GPCoder-ThreadPool-1 Finished.  [monitor] [4/2] Active: 0, Completed: 6, Task: 6, isShutdown: **false**, isTerminated: **false**  [monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: **false**, isTerminated: **false**  [monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: **false**, isTerminated: **false**  [monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: **false**, isTerminated: **false**  [monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: **false**, isTerminated: **false**  [monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: **false**, isTerminated: **false**  [monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: **true**, isTerminated: **true**  [monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: **true**, isTerminated: **true** |

**Một vài lưu ý về sử dụng ExecutorService**

Khi bạn đã thêm tất cả các task cần thiết để thực thi vào executor bạn nên tắt nó bằng phương thức **shutdown()**. Khi bạn gọi phương thức này có nghĩa ExcutorService sẽ từ chối nhận thêm các task, và một khi tất cả các nhiệm vụ đã được thêm vào trước đó đã hòan thành thì Executor sẽ được tắt. Có nghĩa tất cả các task được thêm vào trước khi gọi shutdown() đều sẽ được thực thi, các task thêm sau sẽ bị từ chối (rejected).

Nếu bạn muốn tắt ExecutorService ngay lập tức, bạn có thể gọi phương thức **shutdownNow()**. Điều này sẽ cố gắng ngăn chặn tất cả các nhiệm vụ ngay lập tức và loại bỏ các nhiệm vụ đã được đưa vào Queue nhưng chưa được thực thi. Không có gì đảm bảo về việc tắt các nhiệm vụ đang chạy hoàn toàn, nhưng phương thức này sẽ cố gắng để tắt tất cả chúng.

**Lưu ý:** Sử dụng phương thức **shutdownNow()** cũng tương tự việc sử dụng **interupt()** trong Thread. Để an toàn và chắc chắn hơn khi stop các task đang được thực thi bạn nên thêm vào 1 biến **flag** chung cho các nhiệm vụ, khi bạn stop tất cả Thread sử dụng kèm cả **shutdownNow()** và set giá trị cho biến **flag** để quá trình dừng lại.

**Lời kết:**

Trên đây là những hướng dẫn cơ bản về tạo và sử dụng ThreadPool trong Java. Nếu ứng dụng của bạn cần thực thi nhiều luồng cùng lúc thì bạn nên cố gắng sử dụng ThreadPool để hiệu năng tốt hơn, tiết kiệm thời gian vì không cần phải tạo thread mới cho mỗi task.

Hy vọng bài viết này giúp ích cho các bạn, hẹn gặp lại ở các bài viết tiếp theo.

**Tài liệu tham khảo:**

* <https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice>
* <https://alvinalexander.com/java/jwarehouse/guava/guava/src/com/google/common/util/concurrent/ThreadFactoryBuilder.java.shtml>

**4.6**

20

**Nếu bạn thấy hay thì hãy chia sẻ bài viết cho mọi người nhé!**

SHARES

*Chuyên mục:*[**Multi-Thread**](https://gpcoder.com/category/java-core/multi-thread/)*Được gắn thẻ:*[**Executor Service**](https://gpcoder.com/tag/executor-service/)*,*[**Multithreading**](https://gpcoder.com/tag/multithreading/)*,*[**Thread Pool**](https://gpcoder.com/tag/thread-pool/)

[Đồng bộ hóa các luồng trong Java](https://gpcoder.com/3514-dong-bo-hoa-cac-luong-trong-java/)

[Lập trình đa luồng với Callable và Future trong Java](https://gpcoder.com/3565-lap-trinh-da-luong-voi-callable-va-future-trong-java/)

**Có thể bạn muốn xem:**

* [Lập trình đa luồng với CompletableFuture trong Java 8](https://gpcoder.com/4064-lap-trinh-da-luong-voi-completablefuture-trong-java-8/) **(19/07/2018)**
* [Sử dụng Fork/Join Framework với ForkJoinPool trong Java](https://gpcoder.com/3573-su-dung-fork-join-framework-voi-forkjoinpool-trong-java/) **(06/03/2018)**
* [Thực thi nhiều tác vụ cùng lúc như thế nào trong Java?](https://gpcoder.com/5100-thuc-thi-nhieu-tac-vu-cung-luc-nhu-the-nao-trong-java/) **(01/02/2019)**
* [Vấn đề Nhà sản xuất (Producer) – Người tiêu dùng (Consumer) và đồng bộ hóa các luồng trong Java](https://gpcoder.com/6180-van-de-nguoi-san-xuat-producer-nguoi-tieu-dung-consumer-va-dong-bo-hoa-cac-luong-trong-java/) **(23/09/2019)**
* [Semaphore trong Java](https://gpcoder.com/6187-semaphore-trong-java/) **(18/09/2019)**

**Bình luận**

[](https://facebook.com/GPCoder)

[**GP Coder - Lập trình Java**](https://facebook.com/GPCoder)

Source code đầy đủ các bạn có thể xem trên github nhé:  
<https://github.com/gpcodervn/Java-Tutorial/tree/master/MultiThreadTutorial>